# Respectful Type Converters For Mutable Types

Jeannette M. Wing John Ockerbloom

June 1999

CMU-CS-99-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to *Volume on Foundations of Component Based Systems*, edited by Gary Leavens and Murali Sitaraman

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031 and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

**Keywords:** type converters, mutable types, subtype, object-oriented design, abstraction function, specifications, semantics, Larch, TOM, Y2K.

## Abstract

In converting an object of one type to another, we expect some of the original object's behavior to remain the same, and some to change. How can we state the relationship between the original object and converted object to characterize what information is preserved and what is lost åfter the conversion takes place? We answer this question by introducing the new relation, *respects*, and say that a type converter function $K : A \rightarrow B$ *respects* a type $T$. We formally define *respects* in terms of the Liskov and Wing behavioral notion of subtyping; types $A$ and $B$ are subtypes of $T$.
In previous work [11] we defined *respects* for immutable types A, B, and T; in this paper we extend our notion to handle conversions between mutable types. This extension is non-trivial since we need to consider an object's behavior as it varies over time. We present in detail two examples to illustrate our ideas: one for converting between PNG images and GIF images and another for converting between different kinds of bounded event queues. We also discuss in less detail other real-world applications, namely those inspired by our Typed Object Model (TOM) conversion service built at Carnegie Mellon and by the infamous Year 2000 (Y2K) problem.

# 1 Motivation

The tremendous growth of the Internet and the World Wide Web gives millions of people access to vast quantities of data. While users may be able to retrieve data easily, they may not be able to interpret or display retrieved data intelligibly. For example, when retrieving a Microsoft Word document, without a Microsoft Word program, the user will be unable to read, edit, display, or print it. In general, the type of the retrieved data may be unknown to the retrieving site.

Users and programs cope with this problem by *converting* data from one type to another, e.g., from the unknown type to one known by the local user or program. Thus, to view the Word document, we could convert it to ASCII text or HTML, and then view it through our favorite text editor or browser. A picture in an unfamiliar Windows bitmap type could be converted into a more familiar GIF image type. A mail message with incomprehensible MIME attachments could be converted from an unreadable MIME-encoded type to a text, image, or audio type that the recipient could examine directly. In general, we apply *type converters* on (data) objects, transforming an object of one type to an object of a different type.

## 1.1 What Information Do Type Converters Preserve?

In converting objects of one type to another we expect there to be some relationship between the original object and the converted one. In what way are they similar? The reason to apply a converter in the first place is that we expect some things about the original object to change in a way that we are willing to forgo, but we also expect some things to stay the same. For example, suppose we convert a LaTeX file to an HTML file. We may care to ensure that the raw textual contents of the original LaTeX document are preserved, but not the formatting commands since they do not contribute to the meaning of the document itself; here the preserved information is the underlying semantics of the text contained in the document. Alternatively, if we convert a LaTeX file to a table-of-contents document, we may care to ensure that the number, order, and titles of chapters and sections in the original document are preserved, but not the bulk of the text; here the preserved information is primarily the document's structure.

The question we address in this paper is "How can we characterize what information is preserved by a type converter?" Our answer is given in terms of the behavior of some type $T$. Informally, we say a *converter* $K : A \rightarrow B$ *respects type* $T$ if the original object of type $A$ and the converted object of type $B$ have the same behavior when both objects are viewed as a type $T$ object. That is, from $T$'s viewpoint, the $A$ and $B$ objects look the same. If the converter respects a type, then it preserves that type's observable behavior. This paper formalizes this novel notion of *respectful type converters.*

Our particular formalization of *respects* exploits the *subtype* relationship that holds among types of objects. The Liskov and Wing notion of behavioral subtyping [8] conveniently characterizes semantic differences between types. If $S$ is a subtype of $T$, users of $T$ objects cannot perceive when objects of type $S$ are substituted for $T$ objects. Intuitively, if $K$ respects type $T$, an ancestor of both $A$ and $B$ in the subtype hierarchy, then $T$ captures the behavioral information preserved by $K$.

In our previous work, "Respectful Type Converters" [11] we presented a definition of *respects* for conversions between immutable types only. (It was correspondingly based on a simplified version of Liskov and Wing's definition of subtype.) In this paper, we present an enhanced version of our respects relation that captures important properties of conversions between mutable types. Specifically, for mutable types we say that a conversion *respects* a certain type $T$ if an object with the converted value cannot be distinguished (using $T$'s interface specification) from an object with the original value either at the time of conversion, or by analyzing any future computation on the object. That is, the future subhistory of the new object will not be inconsistent with the expectations raised by the past subhistory of the original object, given the constraints of type $T$.

This paper spells out how to determine whether a given ancestor $T$ in a type hierarchy is respected by a converter $K : A \rightarrow B$ (Figure 1). In general, $A$ and $B$ need not be subtypes of each other; in practice, they are often siblings or cousins in a given type hierarchy. Also, in general, $T$ is not necessarily the least common ancestor of $A$ and $B$.

Here is an example of why $T$ is not just any ancestor of $A$ and $B$. Figure 2 depicts a type family for images. Suppose that the PNG image and GIF image types are both subtypes of a pixel_map type that specifies the colors of the pixels in a rectangular region. GIF images are limited to 256 distinct colors; PNG

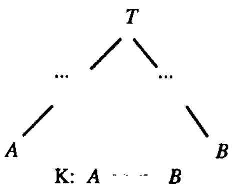

Figure 1: Does Converter $K$ Respect Type $T$?

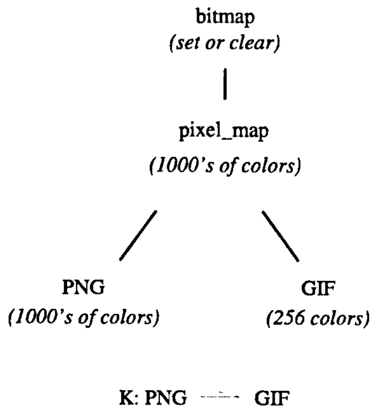

Figure 2: A PNG to GIF converter that does not respect pixel_map might still respect bitmap. Conversely, it is easy to define a GIF to PNG converter that respects both pixel_map and bitmap.

images are not. Assuming the pixel_map type does not have a fixed color limit, then a general converter from PNG images to GIF images would not respect the pixel_map type: it is possible to use pixel_map's interface to distinguish a PNG image with thousands of colors from its conversion to a GIF image with at most 256 colors. On the other hand, suppose pixel_map is in turn a subtype of a more generic bitmap type that simply records whether a graphical element is set or clear. Suppose further that elements in a pixel_map are considered set if they are not black, and clear if they are black. As long as the PNG to GIF converter does not change any non-black color to black (or black to non-black), and otherwise preserves the pixel layout, there is no way for the bitmap interface to distinguish the PNG image from the GIF image that results from the conversion. Here then, the PNG to GIF converter respects the bitmap type.

## 1.2 Typed Object Model Context

At Carnegie Mellon we built a type broker, an instance of Ockerbloom's Typed Object Model (TOM) [9], that provides a type conversion service. Our TOM type broker allows users in a distributed environment to store types and type conversion functions, to register new ones, and to find existing ones. It plays a role similar to that of the type repository in Open Distributed Processing [2] and the conversion manager in System 33 [10]. The kinds of types TOM supports today are different kinds of document types (e.g., Word, LaTeX, PowerPoint, binhex, HTML) and "packages" of such document types (e.g., a mail message that has an embedded postscript file, a tar file, or a zip file). The kinds of conversions TOM supports are off-the-shelf converters like *postscript2pdf* (i.e., AdobeDistiller$^{TM}$), off-the-Web ones like *latex2html*, and some home-grown ones like *powerpoint2html*.

Users can compose available converters to produce an object of a desired target type. For example, to make a fourteen-year old Scribe document available on the Web, the first author used a *scribe2latex* converter to produce a LaTeX file and then an enhanced *latex2html* converter to produce the Web version. This scenario is similar to that described in the previous section: here, the composition of the two converters preserves the semantics of the original Scribe document. Ironically, the LaTeX program failed to run successfully on the intermediate LaTeX file, but for this application, it did not matter; it was the end result—the HTML file—that mattered.

The Web site for the TOM conversion service at Carnegie Mellon is: `http://tom.cs.cmu.edu/`. It supports roughly 100 abstract data types, a few hundred concrete data types, and over 300 type converters (including over 200 meaningful compositions of about 70 primitive converters). As of May 1999, the number of accesses to the TOM conversion service stabilized to 5000 per month, which is an average of 167 per day. Accesses came from over 1000 sites in over 35 countries in six continents from all types of organizations including educational, government, and commercial institutions. One class of the most popular converters are those to unwrap mail messages with embedded files. Another class consists of those which take a document of one type and produce a file of a different type, typically for viewing or printing. The most common source types for converting are mail messages, Microsoft Word, PowerPoint, postscript, LaTeX, and pdf files; the most common target types are Web sections, HTML, text, postscript, and GIF images.

Though our idea of respectful type converters was inspired by our use of TOM in the context of file and document converters, type converters show up in other contexts. Most programming languages have built-in type converters defined on primitive types, for example, *ascii2integer*, *char2string*, and *string2array[char]*. The real world is continually faced with painful, costly, yet seemingly simple conversions: the U.S. Postal System converted five-digit zip codes to five+four-digit zip codes; Bell Atlantic recently added a new area code necessitating the conversion of a large portion of phone numbers in Western Pennsylvania from the 412 area code to 724; payroll processing centers routinely need to convert large databases of employee records whenever extra fields are added to the relevant database schema; and of course, the infamous Year 2000 (Y2K) conversion problem is costing billions of dollars to fix [6].

### 1.3 Roadmap to Rest of Paper

In this paper we formally characterize the notion of when a converter *respects* a type. We first review in Section 2 how we specify types and determine when one type is a subtype of another, borrowing directly from Liskov and Wing's behavioral notion of subtyping [8]. In Section 3 we exploit this notion of subtyping to define the *respects* relation between a converter and a type. In Section 4 we discuss in detail two examples to illustrate converters that do and do not respect types: one example of a type family for PNG and GIF images, and one for bounded event queues. In Section 5 we give a further extension of *respects* that enables us to relate the implementation of types to their specifications; specifically, the abstraction function used to show the correctness of the implementation of abstract types fits neatly into the way we define *respects*. In Section 6 we briefly discuss two "real world" contexts in which our *respects* relation applies: the TOM conversion service and the Y2K problem. We close with a discussion of related work and a summarizing set of conclusions.

## 2 Behavioral Subtyping

The programming language community has come up with many definitions of the subtype relation. The goal is to determine when this assignment

$$x\colon T := E$$

is legal in the presence of subtyping. Once the assignment has occurred, $x$ will be used according to its "apparent" type $T$, with the expectation that if the program performs correctly when the actual type of $x$'s object is $T$, it will also work correctly if the actual type of the object denoted by $x$ is a subtype of $T$.

What we need is a subtype requirement that constrains the behavior of subtypes so that users will not encounter any surprises:

> *No Surprises Requirement*: Properties that users rely on to hold of an object of a type $T$ should hold even if the object is actually a member of a subtype $S$ of $T$.

which guarantees Liskov's *substitutability* principle of subtypes [7]. In their 1994 TOPLAS paper "A Behavioral Notion of Subtyping" Liskov and Wing [8] formalized this requirement in their definition of subtyping. The novel aspect of their subtype definition is the ability to handle mutable types, and in particular, a type's *history* properties.

To provide background for our definition and to make this paper self-contained, we first describe our model of objects and types, then how we specify types, and then we define the subtype relation. These definitions are all taken from the Liskov and Wing paper [8].

## 2.1 Model of Objects, Types, and Computation

We assume a set of all potentially existing objects, *Obj*, partitioned into disjoint typed sets. Each object has a unique identity. A *type* defines a set of *values* for an object and a set of *methods* that provide the only means to manipulate or observe that object.

Objects can be created and manipulated in the course of program execution. A *state* defines a value for each existing object. It is a pair of mappings, an *environment* and a *store*. An environment maps program variables to objects; a store maps objects to values.

$$\begin{aligned} &\mathit{State} = \mathit{Env} \times \mathit{Store} \\ &\mathit{Env} = \mathit{Var} \rightarrow \mathit{Obj} \\ &\mathit{Store} = \mathit{Obj} \rightarrow \mathit{Val} \end{aligned}$$

Given a variable, $x$, and a state, $\rho$, with an environment, $\rho.e$, and store, $\rho.s$, we use the notation $x_\rho$ to denote the value of $x$ in state $\rho$; i.e., $x_\rho = \rho.s(\rho.e(x))$ When we refer to the domain of a state, $dom(\rho)$, we mean more precisely the domain of the store in that state.

We model a type as a triple, $\langle O, V, M \rangle$, where $O \subseteq \mathit{Obj}$ is a set of objects, $V \subseteq \mathit{Val}$ is a set of values, and $M$ is a set of methods. Each method for an object is a *constructor*, an *observer*, or a *mutator*. Constructors of an object of type $\tau$ return new objects of type $\tau$; observers return results of other types; mutators modify the values of objects of type $\tau$. An object is *immutable* if its value cannot change and otherwise it is *mutable*. A type is immutable if all of its objects are; otherwise it is mutable. We allow *mixed methods* where a constructor or an observer can also be a mutator. We also allow methods to signal exceptions; we assume termination exceptions, i.e., each method call either terminates normally or in one of a number of named exception conditions. To be consistent with object-oriented language notation, we write *x.m(a)* to denote the call of method $m$ on object $x$ with the sequence of arguments $a$.

Objects come into existence and get their initial values through *creators*. Unlike other kinds of methods, creators do not belong to particular objects, but rather are independent operations. They are the *class methods*; the other methods are the *instance methods*.

A *computation*, i.e., program execution, is a sequence of alternating states and transitions starting in some initial state, $\rho_0$:

$$\rho_0 \;\; Tr_1 \;\; \rho_1 \;\; \ldots \;\; \rho_{n-1} \;\; Tr_n \;\; \rho_n$$

Each transition, $Tr_i$, of a computation sequence is a partial function on states. A *history* is the subsequence of states of a computation.

Objects are never destroyed: $\forall\; 1 \leq i \leq n \;.\; dom(\rho_{i-1}) \;\subseteq\; dom(\rho_i)$.

## 2.2 Type Specifications

A type specification contains the following information:

- The type's name.
- A description of the set of values over which objects of the type ranges.
- For each of the type's methods:

```
GIF: type

uses GIFImage (gif for G)
for all g: GIF
    invariant $| colorrange(g_\rho) | \leq 256$
    constraint true

  color get_color (i, j: int)
        ensures  $result = overlay(g, i, j)$

  bool set_color (i, j: int; c: color)
        modifies g
        ensures  if  $| colorrange(changepixel(g_{pre}, i, j, c)) | \leq 256$
                 then $c \in colorrange(g_{post}) \wedge g_{post} = changepixel(g_{pre}, i, j, c) \wedge result = true$
                 else $g_{pre} = g_{post} \;\wedge\; result = false$

  frame get_frame (i: int)
        requires $1 \leq i \leq len(g)$
        ensures  $result = g[i]$

end GIF
```

Figure 3: A Larch Type Specification for GIF Images

  - Its name.
  - Its signature, i.e., the types of its arguments (in order), result, and signaled exceptions.
  - Its behavior in terms of pre-conditions and post-conditions.

- A description of the type's history properties.

Figure 3 gives an example of a type specification for GIF images. We give formal specifications, written in the style of Larch [5], but we could just as easily have written informal specifications. Since these specifications are formal we can do formal proofs, possibly with machine assistance like with the Larch Prover [3], to show that a subtype relation holds [12].

The GIFImage Larch Shared Language trait and the **invariant** clause in the Larch interface type specification for GIF images together describe the set of values over which GIF image objects can range. GIF images are sequences of frames where each frame is a bounded two-dimensional array of colors. Appendix A contains all traits used in all examples in this paper, and here in particular, the GIFImage trait and those for frame sequences, frames, colors, etc.

A type invariant constrains the value space for a type's objects. In the GIF example, the type invariant says that a GIF image can have at most 256 different colors. (The *colorrange* function defined in GIFImage returns the range of colors mapped onto by the array.) The predicate $\phi(x_\rho)$ appearing in an **invariant** clause for type $\tau$ stands for the predicate: For all computations $c$, and for all states $\rho$ in $c$:

$$\forall x : \tau \;.\; x \in dom(\rho) \Rightarrow \phi(x_\rho).$$

Whereas an invariant property is a property true of all states of an object, a history property is a property that is true of all sequences of states that result from any computation on that object. For example, a typical history property is that the bound of a finite queue never changes; another example is that the value of an integer counter always monotonically increases. A type constraint in a Larch interface specifies the history properties of the type's objects. The two-state predicate $\phi(x_{\rho_i}, x_{\rho_k})$ appearing in a **constraint** clause for type $\tau$ stands for the predicate: For all computations $c$, and for all states $\rho_i$ and $\rho_k$ in $c$ such that $i < k$:

$$\forall x : \tau \; . \; x \in dom(\rho_i) \Rightarrow \phi(x_{\rho_i}, x_{\rho_k})$$

Note that we do not require that $\rho_k$ be the immediate successor of $\rho_i$ in $c$. The GIF example has the trivial constraint *true*. In Section 4.2 we will give examples of types with non-trivial constraints.

The **requires** and **ensures** clauses in the Larch interface specification state the methods' pre- and post-conditions respectively. To be consistent with the Liskov and Wing paper and the Larch approach, pre-conditions are single-state predicates and post-conditions are two-state predicates. The **modifies** clause states that the values of any objects it does *not* list do not change; the values of those listed may possibly change. The absence of a **requires** clause stands for the pre-condition *true*. The absence of a **modifies** clause means that the method cannot change the values of any objects.

The *get_color* method returns the color of the (i, j)th array element of $g$. The *overlay* function defined in GIFImage returns the color value of the $(i, j)$th array element of the last frame in the sequence that gives a value for $(i, j)$; otherwise, it returns BLACK, a distinguished color value, introduced in the ColorLiterals trait. For example, if there are three frames in the frame sequence and for a given $(i, j)$, the first frame maps the array element to BLACK, the second to RED, and the third does not map $(i, j)$ to any color (because it is not within its bounds), then RED is returned. The *set_color* method modifies the GIF object $g$ by changing the final color of pixel $(i, j)$ to $c$, and returns true, if the change would not make the resulting GIF have more than 256 colors. Otherwise it leaves the GIF object unchanged and returns false. The *get_frame* method returns the $i$th frame of the GIF object's value.

To ensure that the specification is *consistent*, the specifier must show that each creator for the type $\tau$ establishes $\tau$'s invariant, and that each of $\tau$'s methods both preserves the invariant and satisfies the constraint. These are standard conditions and their proofs are typically straightforward [8].

## 2.3 The Subtype Relation

The subtype relation is defined in terms of a checklist of properties that must hold between the specifications of the two types, $\sigma$ and $\tau$. Since in general the value space for objects of type $\sigma$ will be different from the value space for those of type $\tau$ we need to relate the different value spaces; we use an *abstraction function*, $\alpha$, to define this relationship. Also since in general the names of the methods of type $\sigma$ can be different from those of type $\tau$ we need to relate which method of $\sigma$ corresponds to which method of $\tau$; to define this correspondence we use a *renaming* map, $\nu$, that maps names of methods of $\sigma$ to names of methods of $\tau$. (In a programming language like Java, this is just the identity map, as realized through method overloading.)

$\sigma$ is a *subtype* of $\tau$ if the following three conditions hold (informally stated):

1. The abstraction function respects the invariants. If the subtype invariant holds for any subtype value, $s$, then the supertype invariant must hold for the abstracted supertype value $\alpha(s)$.

2. Subtype methods preserve the supertype methods' behavior. If $m$ is a subtype method then let $n$ be the corresponding $\nu(m)$ method of the supertype.

   - Signature rules.
     - Arguments to $m$ are contravariant to the corresponding arguments to $n$; $m$'s result is covariant to the result of $n$.
     - Any exception signaled by $m$ is contained in the set of exceptions signaled by $n$.
   - Methods rules. (These are completely analogous to the contra/covariant signature rules.)
     - $n$'s pre-condition implies $m$'s (under the abstraction function).
     - $m$'s post-condition implies $n$'s (under the abstraction function).

3. The history constraints of the supertype are preserved in the subtype. (The subtype can add additional constraints, but its constraints must be at least as strict as its supertype's.)

The formal definition of the subtype relation, $<$, is given in Figure 4 [1]. It relates two types, $\sigma$ and $\tau$, each of whose specifications we assume are consistent. In the methods rules, since $x$ is an object of type $\sigma$,

[1] It is Liskov and Wing's "constraint" based subtype definition, and is taken from Fig. 4 of [8].

its value ($x_{pre}$ or $x_{post}$) is a member of $S$ and therefore cannot be used directly in the predicates about $\tau$ objects (which are in terms of values in $T$). The abstraction function $\alpha$ is used to translate these values so that the predicates about $\tau$ objects make sense.

---

DEFINITION OF THE SUBTYPE RELATION, $<$: $\sigma = \langle O_\sigma, S, M\rangle$ is a *subtype* of $\tau = \langle O_\tau, T, N\rangle$ if there exists an abstraction function, $\alpha : S \rightarrow T$, and a renaming map, $\nu : M \rightarrow N$, such that:

1. The abstraction function respects invariants:

   - *Invariant Rule.* $\forall s : S \;.\; I_\sigma(s) \Rightarrow I_\tau(\alpha(s))$

   $\alpha$ may be partial, need not be onto, but can be many-to-one.

2. Subtype methods preserve the supertype methods' behavior. If $m_\tau$ of $\tau$ is the corresponding renamed method $m_\sigma$ of $\sigma$, the following rules must hold:

   - *Signature rule.*
     - *Contravariance of arguments.* $m_\tau$ and $m_\sigma$ have the same number of arguments. If the list of argument types of $m_\tau$ is $a_i$ and that of $m_\sigma$ is $b_i$, then $\forall i \;.\; a_i < b_i$.
     - *Covariance of result.* Either both $m_\tau$ and $m_\sigma$ have a result or neither has. If there is a result, let $m_\tau$'s result type be $a$ and $m_\sigma$'s be $b$. Then $b < a$.
     - *Exception rule.* The exceptions signaled by $m_\sigma$ are contained in the set of exceptions signaled by $m_\tau$.
   - *Methods rule.* For all $x : \sigma$:
     - *Pre-condition rule.* $m_\tau.pre[\alpha(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
     - *Post-condition rule.* $m_\sigma.post \Rightarrow m_\tau.post[\alpha(x_{pre})/x_{pre}, \alpha(x_{post})/x_{post}]$

3. Subtype constraints ensure supertype constraints.

   - *Constraint rule.* For all computations $c$, and all states $\rho_i$ and $\rho_k$ in $c$ where $i < k$, for all $x : \sigma$:
     $C_\sigma \Rightarrow C_\tau[\alpha(x_{\rho_i})/x_{\rho_i}, \alpha(x_{\rho_k})/x_{\rho_k}]$

Figure 4: Definition of the Subtype Relation

---

Why does this subtype relation guarantee that the No Surprises Requirement holds? Recall that the Requirement refers informally to "properties." This definition of subtype guarantees that certain properties of the supertype—those stated explicitly or provable from a type's specification—are preserved by the subtype. The first condition directly relates the invariant properties. The second condition relates the behaviors of the individual methods, and thus preserves any observable behavioral property of any program that invokes those methods. The third condition relates the overall histories of objects, guaranteeing that the possible histories in the subtype specification are also possible histories in the supertype specification.

Figure 5 gives a type specification for pixel_map, which is a supertype of both GIF and PNG. To show that GIF is a subtype of pixel_map (Figure 2), we define the following abstraction function:

$\alpha_G^{PM} : \mathrm{G} \rightarrow \mathrm{PM}$
$\forall i, j : \mathrm{Integer} \;.\; \alpha_G^{PM}(g)[i, j] = overlay(g, i, j)$

Using this abstraction function, the proofs that the invariant, signature, methods, and constraint rules either are straightforward to show or trivially hold. The only noteworthy aspect of pixel_map's specification is the nondeterminism specified for its *set_color* method, which is more liberal than that for both GIF images and PNG images (as we will see in Section 4). A call to *set_color* can always either fail (making no change) or succeed (possibly adding a new color to the pixel_maps's color range). We exploit this nondeterminism later in our proofs.

pixel_map: **type**

**uses** PixelMap (pixel_map **for** PM)<br>
**for all** $p$: pixel_map

  **invariant** *true*<br>
  **constraint** *true*

  color *get_color* ($i$, $j$: int)<br>
    **ensures** $result = p[i, j]$

  bool *set_color* ($i$, $j$: int; $c$: color)<br>
    **modifies** $p$<br>
    **ensures** $(result = false \wedge p_{pre} = p_{post}) \vee$<br>
    $(result = true \wedge c \in colorrange(p_{post}) \wedge c = p_{post}[i, j] \wedge$<br>
    $\forall k, l : Integer.(k \neq i \vee l \neq j) \Rightarrow p_{pre}[k, l] = p_{post}[k, l])$

**end** pixel_map

Figure 5: A Larch Type Specification for Pixel Maps

# 3 Respects

## 3.1 Definition of Respectful Type Converter

Suppose we have two types $A = \langle O_A, V_A, M_A \rangle$ and $B = \langle O_B, V_B, M_B \rangle$. A converter, $K$, is a partial function from $V_A$ to $V_B$. Thus when we say that a converter maps from type $A$ to type $B$ we mean more precisely that it maps the value space of type $A$ to the value space of type $B$; for notational convenience, we continue to write the signature of $K$ as $A \rightarrow B$. To ensure the converter is consistent with $B$'s specification, the specifier should show that the values of $V_B$ to which $K$ maps satisfy $B$'s type invariant.

Let $T$ be a type that is a common ancestor of $A$ and $B$ in a given type hierarchy. $T$ is a supertype of both $A$ and $B$. Then there exist ancestor types, $A_1 \ldots A_n$, between $A$ and $T$ such that there exist the following abstraction functions:

$$\alpha_0 : A \rightarrow A_1$$
$$\ldots$$
$$\alpha_i : A_i \rightarrow A_{i+1}$$
$$\ldots$$
$$\alpha_n : A_n \rightarrow T$$

Assuming for all $1 \leq i \leq n$ . $dom(\alpha_i) \subseteq ran(\alpha_{i-1})$, let $\alpha$ be the functional composition of $\alpha_i$:

$$\alpha = \alpha_n \circ \ldots \circ \alpha_0$$

For $B$ we similarly define $B_i, \beta_i$, and $\beta$ for $1 \leq i \leq m$. Figure 6 illustrates these constructs.

Figure 7 gives the definition of the *respects* relation for a converter $K : A \rightarrow B$ and type $T$. The first two conditions (under **Methods**) state that the original value and the converted value are indistinguishable when viewed through the methods (in particular, the observers) of type $T$. Let $a$ stand for $y_{pre}$ used in the definition (the value of the object of type A before the call to T's method $m$). Then the first condition requires that $m$'s pre-condition holds for $a$'s abstraction under $\alpha$ iff it holds for the converted value of $a$ abstracted under $\beta$. Thus from $T$'s viewpoint, if $m$ is defined for $A$'s values, it should be defined for $B$'s values, and vice versa. The second condition requires that $m$'s post-condition holds for $a$'s abstracted value under $\alpha$ iff it holds for the converted value of $a$ abstracted under $\beta$. Thus, given that $m$ is defined, then its observed state must be the same for $A$'s values and $B$'s values from $T$'s viewpoint.

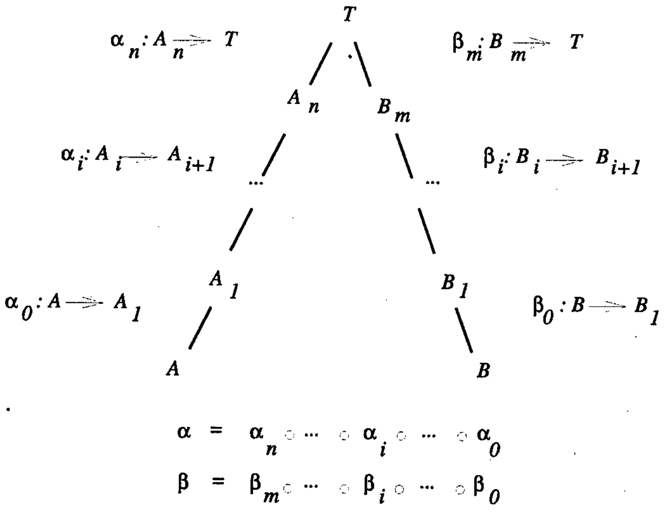
T
$\alpha_n : A_n \rightarrow T$
$\beta_m : B_m \rightarrow T$
$A_n$
$B_m$
$\alpha_i : A_i \rightarrow A_{i+1}$
$\beta_i : B_i \rightarrow B_{i+1}$
...
...
$A_1$
$B_1$
$\alpha_0 : A \rightarrow A_1$
$\beta_0 : B \rightarrow B_1$
A
B
$\alpha = \alpha_n \circ \cdots \circ \alpha_i \circ \cdots \circ \alpha_0$
$\beta = \beta_m \circ \cdots \circ \beta_i \circ \cdots \circ \beta_0$

Figure 6: Two Compositions of Abstraction Functions

DEFINITION OF RESPECTS RELATION: Let $K : A \rightarrow B$ be a partial function mapping values of type $A$ to values of type $B$. Let $T$ be an ancestor of both $A$ and $B$ in a given type hierarchy; $\alpha$ and $\beta$ are defined in text and depicted in Figure 6. Then *converter* $K$ *respects* $T$ if the following conditions hold:

- **Methods**: For each method $m$ of $T$, and for all objects $x : T$, $y : A$, and $z : B$, and for all subcomputations $y_{pre}\ \ m_A\ \ y_{post}$ and $z_{pre}\ \ m_B\ \ z_{post}$, such that $m_A$ and $m_B$ are $A$'s and $B$'s corresponding methods of $m$, $y_{pre} \in dom(K)$, and $K(y_{pre}) = z_{pre}$:

  1. $m.pre[\alpha(y_{pre})/x_{pre}] \Leftrightarrow m.pre[\beta(z_{pre})/x_{pre}]$ and
  2. $m.post[\alpha(y_{pre})/x_{pre}, \alpha(y_{post})/x_{post}] \Leftrightarrow m.post[\beta(z_{pre})/x_{pre}, \beta(z_{post})/x_{post}]$

- **Constraint**: For all integers $i$, $j$, and $k$, where $0 \leq i \leq j \leq k$, all histories $\rho_0 ... \rho_k$ and $\psi_0 ... \psi_k$, and for all objects $x : T$, $y : A$, and $z : B$ such that $y_{\rho_j} \in dom(K)$ and $K(y_{\rho_j}) = z_{\psi_j}$:
  $C_T[\alpha(y_{\rho_i})/x_{\rho_i}, \beta(z_{\psi_k})/x_{\rho_k}]$

Figure 7: Definition of the Respects Relation

The last condition (labeled **Constraint**) requires that $T$'s constraints between any two points of any history must be the same for an unconverted object of type A as for a converted object of type B. This is trivially true for two points before the conversion and for two points after the conversion; the condition more generally handles the case where one point, $\rho_i$, is before the conversion, and one point, $\rho_k$, is after the conversion, where the point of conversion is $\rho_j$. Thus, from $T$'s viewpoint, the converted object's later states are consistent with the earlier states of the original object, given the history properties of $T$. To put it another way, now let $a$ stand for $y_{\rho_j}$ used in the definition. If T's history constraint holds, then an observer cannot tell that the object with the new value after conversion, $K(a)$, is any different from the object with the original value, $a$, before conversion.

These conditions together guarantee that $T$'s behavior is preserved by the conversion of objects of type $A$ to those of type $B$. Informally, $T$ cannot distinguish between an object with the original value and an object with the converted value, even when taking the subsequent histories of the objects into account. Thus $K$ *respects* $T$.

**Claim 1** *If a and $K(a)$ abstractly map to the same value in $T$, i.e., $\alpha(a) = \beta(K(a))$ for all a in the domain of $K$, then the respects relation trivially follows.*

This special case is often useful in proofs that a converter respects a type, as we will see Section 4.

### 3.2 Discussion of Definition

The definition of the respects relation has a similar structure to the definition of the subtype relation. Like the subtype relation, the respects relation includes methods and constraint rules. The **Methods** rules show that the observable state of the original object and the object with the converted value are indistinguishable (from the respected type's point of view) at the time of conversion. The **Constraint** rule shows that the history of the object with the converted value is consistent with the past history of the original object. Unlike the subtype relation, the respects relation does not need an invariant rule. Since both types $A$ and $B$ are subtypes of the respected type $T$, *all* objects of those types must conform to the invariants of $T$, by the definition of subtype, so restating the invariant rule here for particular $A$ and $B$ objects would be superfluous.

We considered various alternate definitions of the respects relation. One of them explicitly modeled a computation involving object $z : B$, and required it to correspond exactly to a computation involving object $y : A$, in rules like the ones above. Another did not explicitly consider histories or computations at all, but simply compared values of an object $y : A$ to values of a corresponding object $z : B$, and made sure the converted values of $z$ matched the original values of $y$ at every state. The first alternative was more complex than necessary; the second, too simplistic. Moreover, both failed to accommodate cases where the history of an object with a converted value diverges from that of an object with an original value, a situation we want (and need) to allow. (In the next section, we will see examples of this phenomenon. One common case occurs where one of the types in question is less constrained than the other.) The history of the original object and that of the converted object are allowed to diverge after the point of conversion, as long as both histories are consistent with the past history of the original object, with respect to the respected type's constraints. As long as this consistency property holds, there should be "no surprises" from the respected type's point of view after a conversion.

## 4 Two Examples

The first example shows how our definition handles mutable methods, and hence mutable types. The second example goes one step further and shows how we handle history properties as specified in non-trivial constraint clauses.

### 4.1 PNG and GIF Example Revisited

Let us look at the PNG to GIF example more carefully. First, we give the type specification for PNG images and an abstraction function that enables us to argue that PNG is a subtype of pixel_map. Then we consider converters between PNG and GIF, to argue that no total converter from PNG to GIF respects pixel_map, but that some converters from GIF to PNG do.

The type specification for PNG images is given in Figure 8. Note we have a nontrivial application of the renaming map, $\nu$, where $\nu(\textit{get\_corrected\_color}) = \textit{get\_color}$, and $\nu(\textit{set\_corrected\_color}) = \textit{set\_color}$. (Only some of the methods for PNG have corresponding supertype methods; the rest are left unmapped by $\nu$.)

We are always allowed to set a PNG image's pixels to new colors, with no limit on the total number of colors in the image; this freedom follows from the trivial type invariant and *set_corrected_color*'s specification. In contrast, we are allowed to set a GIF image's pixels to new colors only when the total number of colors does not exceed 256. The nondeterminism in the pixel_map supertype (Figure 5) accommodates both subtype specifications. In particular, for PNG images, the color range grows as more colors are added, so that *set_corrected_color* always successfully sets a color, if a coordinate is set within the PNG image's bounds. Moreover, although the pixel_map supertype does not have any concept of coordinate boundaries, its *set_color* method can fail for any reason, thus accommodating the behavior of PNG's *set_corrected_color* in the case that the coordinates are out of bounds.

```
PNG: type

uses PNGImage (PNG for P)
for all $p$: PNG
        invariant $true$
        constraint $true$

    color get_uncorrected_color ($i$, $j$: int)
                requires  $inframe(p, i, j)$
                ensures   $result = p[i, j]$

    gamma get_gamma ()
                ensures   $result = gamma(p)$

    int get_xmin ()
                ensures   $result = xmin(p)$

    ... and similarly for get_xmax, get_ymin, and get_ymax ...

    color get_corrected_color ($i$, $j$: int)
                ensures   if $inframe(p, i, j)$
                          then $result = gc(p[i, j], gamma(p), \mathrm{STDG})$
                          else $result = \mathrm{BLACK}$

    bool set_corrected_color ($i$, $j$: int; $c$: color)
                modifies  $p$
                ensures   $same\_bounds\_and\_gamma(p_{pre}, p_{post}) \wedge$
                          if $inframe(p, i, j)$
                          then $c = gc(p_{post}[i, j], gamma(p_{post}), \mathrm{STDG}) \wedge$
                                $c \in colorrange(p_{post}) \wedge$
                                $\forall k, l : Integer.(k \neq i \vee l \neq j) \Rightarrow p_{pre}[k, l] = p_{post}[k, l])) \wedge$
                                $result = true$
                          else $p_{pre} = p_{post} \ \wedge \ result = false$

end PNG
```

Figure 8: A Larch Type Specification for PNG Images

PNG images differ from pixel_map objects in two ways: (1) they are framed and (2) associated with each PNG object, $p$, is a "gamma" value, denoted *gamma(p)*, used in a *gamma correction* function, *gc*. The gamma correction function corrects for differences among monitors; some are dimmer than others and thus have different color balances. We abstract from the intricacies of gamma correction functions; for our purposes here, they take as arguments a color, an input gamma factor, an output gamma factor, and return a color. The constant, STDG, is the standard gamma value for normal monitors.

We define the following abstraction function to show that PNG is a subtype of pixel_map:

$\alpha_P^{PM}$ : P → PM

$$\forall i, j : \text{Integer} \ . \ \alpha_P^{PM}(p)[i,j] = \begin{cases} gc(p[i,j], gamma(p), STDG) & \text{if } xmin(p) \le i \le xmax(p) \wedge \\ & \quad ymin(p) \le j \le ymax(p) \\ \text{BLACK} & \text{otherwise} \end{cases}$$

Consider a converter, $K$ : P → G, that maps values of PNG images to GIF values.

**Claim 2** *There is no such converter that respects pixel_map, if the converter is defined for PNG images of more than 256 colors.*

*Proof: A simple counting argument suffices. First we show that for a given PNG value, p, where* $| \ colorrange(p) \ | = n$ *and* $n > 256$*, its abstracted pixel_map value,* $\alpha_P^{PM}(P)$*, also has at least 256 colors. From the abstraction function,* $\alpha_P^{PM}(p)[i,j] = gc(p[i,j], gamma(p), STDG)$*, we know that every array element of p maps to some array element of* $\alpha_P^{PM}(p)$*. Furthermore, if two array elements in p have different colors, so do the corresponding cells in* $\alpha_P^{PM}(p)$*. To prove this, we show that if two gamma corrected colors are the same, then the original colors,* $c_1$ *and* $c_2$*, also have to be the same, i.e.,*

*Suppose*

1. $gc(c_1, gamma(p), STDG) = gc(c_2, gamma(p), STDG)$

*By the "transitivity" and "reflexivity" properties of gamma correction functions (see Appendix A), we know that*

2. $gc(gc(c_1, gamma(p), STDG), STDG, gamma(p)) = c_1$

*By substitution in line 1, we get*

3. $gc(gc(c_2, gamma(p), STDG), STDG, gamma(p)) = c_1$

*Yielding*

4. $c_2 = c_1$

*So if there are* $n > 256$ *colors in p then there are also at least n colors in* $\alpha_P^{PM}(p)$*.*

*Next we show that the conversion of p to a valid GIF image value* $K(p)$ *would cause the GIF value to be observably different from the PNG value, when viewed through* pixel_map*'s interface.* $K(p)$ *can have a maximum of 256 colors by the type invariant of GIF image. Furthermore, the abstraction mapping of* $K(p)$ *to a pixel_map value,* $\alpha_G^{PM}(K(p))$*, cannot add any colors to* $colorrange(K(p))$ *(except for BLACK), since we see from the definition of* $\alpha_G^{PM}$*, and hence by the definition of overlay, that every* $c \in colorrange(\alpha_G^{PM}(K(p)))$ *is either BLACK or one of the colors used in one of the frames of* $K(p)$*. Therefore, there exists some c such that* $c \in colorrange(\alpha_P^{PM}(p))$ *and* $c \ni colorrange(\alpha_G^{PM}(K(p)))$*. So there exists some i and j such that the result of calling* pixel_map*'s observer,* get_color *with arguments i and j will differ between a call on the original PNG image from a call on the converted GIF image, i.e.,* $\alpha_P^{PM}(p)[i,j] \neq \alpha_G^{PM}(K(p))[i,j]$ *Therefore, the converter cannot respect pixel_map.* □

It is possible, however, to have a converter from GIF images to PNG that respects the pixel_map type.

**Claim 3** *There exist converters from GIF to PNG that respect pixel_map.*

*Proof: By existence. Here is a converter from GIF to PNG:*

$K : G \rightarrow P$
$K(g) = p$ *where*
$xmin(p) = min(\{xmin(g[i]) \mid 0 \le i < len(g)\}) \ \wedge$
$xmax(p) = max(\{xmax(g[i]) \mid 0 \le i < len(g)\}) \ \wedge$
$ymin(p) = min(\{ymin(g[i]) \mid 0 \le i < len(g)\}) \ \wedge$
$ymax(p) = max(\{ymax(g[i]) \mid 0 \le i < len(g)\}) \ \wedge$

$$gamma(p) = STDG \wedge$$
$$\forall i, j : \text{Integer} \ . \ p[i, j] = overlay(g, i, j)$$

*Composing our converter with our abstraction function from PNG to pixel_map, for an original gif value, g, we get an abstracted converted pixel_map value, pm, such that*

$$\forall i, j : \text{Integer} \ . \ pm[i, j] = \begin{cases} gc(overlay(g, i, j), STDG, STDG) & \text{if } min(\{xmin(g[k]) \mid 0 \leq k < len(g)\}) \leq i \wedge \\ & i \leq max(\{xmax(g[k]) \mid 0 \leq k < len(g)\}) \wedge \\ & min(\{ymin(g[k]) \mid 0 \leq k < len(g)\}) \leq j \wedge \\ & j \leq max(\{ymax(g[k]) \mid 0 \leq k < len(g)\}) \\ BLACK & otherwise \end{cases}$$

*By the "reflexivity" property of gamma correction functions, we know that $gc(c, g, g) = c$. Furthermore, we know that from the definition of overlay that when $i$ and $j$ are beyond the bounds of any frames in a frameset, $overlay(g, i, j)$ is BLACK. In the definition above, whenever $i$ or $j$ are outside the respective minima or maxima, then $(i, j)$ is outside any frame in the GIF. Therefore the definition above simplifies to*

$$\forall i, j : \text{Integer} \ . \ pm[i, j] = overlay(g, i, j)$$

*which is exactly the same abstraction function as is used to map the original GIF to a pixel map. The abstracted values are identical, so by Claim 1 (made at the end of Section 3.1), the conversion respects pixel_map.* □

Our assertion above may seem to go against our intuition, when we consider the further histories of the original GIF and the converted PNG. After all, the same sequence of mutations called at the pixel_map level can cause the histories of the original object and the converted object to diverge. Indeed, our type specifications for GIF and PNG mandate that the histories sometimes *must* diverge. Consider, for instance, an object $o$ which at some state $\rho$ has exactly 256 colors. Attempting to call *set_color* (within appropriate x and y bounds) with a 257th color *must* fail for the original GIF object, since it has a maximum of 256 colors. However, it *must* succeed for the converted PNG object, since *set_color* as defined on PNG images cannot fail if it is called within the x and y bounds of the image. From this point on, then, the histories of the original GIF image and the converted PNG image diverge.

For our conversion to respect pixel_map, however, it suffices that the GIF image and the converted PNG image, at the time of conversion, have identical *possible* futures from pixel_map's perspective. That is, we should not be able to tell, given only the mutations and requirements of pixel_map, that the object with the original value and the object with the converted value were different at the point of conversion. As long as this is true, programs expecting to operate on a pixel_map object will not encounter surprises if the object they operate on had been converted from a GIF to a PNG image.

Our pixel_map type has only one mutator, *set_color*. All we know about *set_color* is that any attempt to add a new color to a given pixel_map might succeed or might fail. Either outcome is possible from pixel_map's perspective, no matter how many colors are in a pixel map at a given time. So, from pixel_map's perspective, any sequence of pixel_map method calls on both an original GIF object and a converted PNG object will have the same possible future observed behaviors. Since the possible future histories of the PNG and GIF objects look the same from pixel_map's point of view, there will be no surprises when converting from GIF to PNG, if one assumes only the behavior specified in pixel_map. Appendix B contains a formal proof of this argument.

## 4.2 Event Queues

The types in the previous example had invariants but no history constraints. In this section's example of an event queue type family (Figure 9), we look at a conversion between constrained types, and show which common supertypes the conversion respects, and which it does not.

At the root of the type hierarchy, we have an eventqueue type that models buffered event queues. We represent a value of an eventqueue object, $q$, as a pair, [*items, bound*], of a set of the buffered items and a bound. Events in the queue must be inserted in increasing timestamp order. The size of the queue buffer is bounded, but the bound is not directly readable or writable by the eventqueue type. New events (if they have appropriate timestamps) can be inserted into the queue unless the number of items already in the queue

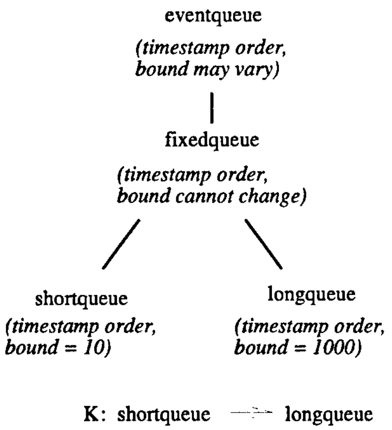
eventqueue
(timestamp order, bound may vary)
fixedqueue
(timestamp order, bound cannot change)
shortqueue
(timestamp order, bound = 10)
longqueue
(timestamp order, bound = 1000)
K: shortqueue → longqueue

Figure 9: A Queue Hierarchy

```
eventqueue = **type**

**uses** EventQueue (eventqueue **for** $Q$)
**for all** $q$: eventqueue
      **invariant** $len(q_\rho.items) \leq q_\rho.bound$
      **constraint** $timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items))$

    bool $insert$ ($e$: event)
                  **requires** $timestamp(last(q_{pre}.items)) < timestamp(e)$
                  **modifies** $q$
                  **ensures** $len(q_{post}.items) \leq q_{post}.bound \land$
                  **if** $len(q_{pre}.items) < q_{post}.bound$
                              **then** $q_{post}.items = add(e, q_{pre}.items) \land result = true$
                              **else** $q_{post}.items = q_{pre}.items \land result = false$

    event $remove$ ( )
                  **requires** $q_{pre}.items \neq empty$
                  **modifies** $q$
                  **ensures** $q_{post}.items = tail(q_{pre}.items) \land result = head(q_{pre}.items) \land$
                              $len(q_{post}.items) \leq q_{post}.bound$

    int $size$ ( )
                  **ensures** $result = len(q_{pre}.items)$

**end** eventqueue
```

Figure 10: A Type Specification for Event Queues

```
fixedqueue = type

uses EventQueue (fixedqueue for $Q$)
for all $q$: fixedqueue
        invariant $len(q_\rho.items) \leq q_\rho.bound$
        constraint
          $timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items)) \land q_{\rho_i}.bound = q_{\rho_k}.bound$

    bool insert (e: event)
                requires $timestamp(last(q_{pre}.items)) < timestamp(e)$
                modifies $q$
                ensures $q_{pre}.bound = q_{post}.bound \land$
                        if $len(q_{pre}.items) < q_{post}.bound$
                        then $q_{post}.items = add(e, q_{pre}.items) \land result = true$
                        else $q_{post}.items = q_{pre}.items \land result = false$

    event remove ( )
                requires $q_{pre}.items \neq empty$
                modifies $q$
                ensures $q_{post}.items = tail(q_{pre}.items) \land result = head(q_{pre}.items) \land$
                        $q_{post}.bound = q_{pre}.bound$


    subtype of eventqueue
                $\forall q : Q \;.\; \alpha(q) = q$

end fixedqueue
```

Figure 11: A Type Specification for Fixed Queues

```
shortqueue = type

uses EventQueue (shortqueue for $Q$)
for all $q$: shortqueue
        invariant $len(q_\rho.items) \leq q_\rho.bound \land q_\rho.bound = 10$
        constraint
          $timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items)) \land q_{\rho_i}.bound = q_{\rho_k}.bound$

    subtype of fixedqueue
                $\forall q : Q \;.\; \alpha(q) = q$

end shortqueue
```

Figure 12: A Type Specification for Short Queues

longqueue = **type**

**uses** EventQueue (longqueue **for** $Q$)

**for all** $q$: longqueue

**invariant** $len(q_\rho.items) \leq q_\rho.bound \wedge q_\rho.bound = 1000$

**constraint**

$timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items)) \wedge q_{\rho_i}.bound = q_{\rho_k}.bound$

**subtype of** fixedqueue

$\forall q : Q \, . \, \alpha(q) = q$

**end** longqueue

Figure 13: A Type Specification for Long Queues

is equal to (or greater than) the bound. The eventqueue type also has the overall constraint that the event at the head of the queue at a state $\rho_i$ has to have a timestamp less than or equal to the event at the head of the queue at any later state $\rho_k$. The constraint, however, does not require that its bound be fixed, so the bound can vary over time. (The specification subtly allows this possible mutation since the *insert* and *remove* methods each has a **modifies** clause; though no mention of changing the queue's bound is made in either of their post-conditions, the presence of the **modifies** clause gives permission to implementors to change that part of the queue's value and simultaneously warns callers that they cannot rely on that part of the queue's value to remain the same. Even more subtly, the condition in the **if** . . . **then** . . . **else** clause for *insert* compares the length of the buffer of the queue's pre-state with the bound of the queue's *post*-state to account for the possibility of the bound changing as a side effect of calling *insert*.)

Let fixedqueue (Figure 11) be a subtype of eventqueue which adds the further constraint that the buffer bound cannot change. Again, there are no methods to read the bound directly. Two subtypes of fixedqueue, shortqueue and longqueue, further specify that the bound is fixed to be 10 and 1000 items, respectively (Figures 12 and 13). The **subtype** clause in a type specification includes an abstraction function, $\alpha$, that relates subtype values to supertype values. Implicitly the clause requires that the subtype provides all methods of its supertype; any method not renamed or redefined is "inherited" from its supertype as is. For example, shortqueue's *insert* does not redefine fixedqueue's *insert*, but fixedqueue's does redefine eventqueue's.

**Claim 4** *There is no conversion from shortqueue to longqueue, either partial or total, that respects fixedqueue.*

*Proof: Consider a shortqueue object s in the domain of the conversion. By the definition of shortqueue, its bound must be 10. Suppose a conversion is made of $s_{\rho_j}$, yielding a longqueue value $l_{\psi_j}$. By the definition of the longqueue type, the bound of the converted object must be 1000. Now consider $s_{\rho_i}$ prior to the conversion and the value $l_{\psi_k}$ after the conversion. The constraint of fixedqueue is violated, since $s_{\rho_i}.bound \neq l_{\psi_k}.bound$, and fixedqueue's constraint prohibits the bound from changing between states. The* **Constraint** *condition of the definition of respects does not hold. Hence, the conversion cannot respect fixedqueue.* □

The histories of the converted object show the failure of the conversion to respect fixedqueue. While it is possible to define a simple conversion from shortqueue to longqueue that contains exactly the same items, the conversion of its bound (from 10 to 1000) changes possible future behaviors of the longqueue object in ways not consistent with the fixedqueue constraints. Suppose, for instance, that we have a program that fills up a queue buffer to determine its size, and uses this information to allocate a fixed-size buffer of its own to store items pulled off the queue. At some later point, after more items have been added to the queue, the program empties the queue items into its own fixed-size buffer. If the queue bound has been increased by a conversion, the program may overflow its previously-allocated buffer, causing a crash or other errors.

To illustrate the incongruity above, we must track the behavior of the original object and the converted object over time, through the point of conversion. It is not enough simply to look at each state and to compare the original object's value and the converted object's value in that same state. In the queue example, the longqueue type is less constrained than the shortqueue type, and so some possible longqueue values are outside the range of any converter on shortqueue values. Once a conversion takes place, we need to reason about the object in terms of its longqueue values; moreover, it would be ill-defined in subsequent states to compare its longqueue value to any shortqueue value. However, we can allow such non-surjective converters as long as the converted object's value does not violate constraints of the respected type.

For instance, it is possible to convert from shortqueue to longqueue in a way that respects the more general eventqueue type.

**Claim 5** *There is a total conversion from shortqueue to longqueue that respects eventqueue.*

*Proof: By existence. Here is such a converter:*

$K : Q \to Q$
$K(q) = [q.items, 1000]$

*To see whether this conversion respects eventqueue, we first check the method rules. The pre- and post-conditions of the methods* remove *and* size, *and the pre-condition of the method* insert, *depend only on the items portion of the queue value, which is the same for both a shortqueue object s and a longqueue object l at the time of conversion. The* insert *method's post-condition depends in part on the post-state of bound, but since eventqueue allows bound to change on* insert, *eventqueue's specification permits either s or l's bound to change, and hence allows the operation to insert an item or not for either object. (While s and l will in fact behave differently based on the more constrained specifications of shortqueue and longqueue, the eventqueue specification itself cannot be used to tell that anything changed in the conversion.)*

*The only constraint of eventqueue is that the timestamp at the head of the queue not decrease over time. Let $s_{\rho_i}$ be any value of s before the conversion, $s_{\rho_j}$ be the value of s at the time of conversion, $l_{\psi_j}$ be the value of l at the time of conversion, and $l_{\psi_k}$ be any value of l after the conversion. Since shortqueue and longqueue are both subtypes of eventqueue, which includes the history constraint, we know that*

*1.* $timestamp(head(s_{\rho_i}.items)) \leq timestamp(head(s_{\rho_j}.items))$ *and*
*2.* $timestamp(head(l_{\psi_j}.items)) \leq timestamp(head(l_{\psi_k}.items))$
*Furthermore, by the definition of our conversion,*
*3.* $s_{\rho_j}.items = l_{\psi_j}.items$
*so therefore*
*4.* $timestamp(head(s_{\rho_j}.items)) = timestamp(head(l_{\psi_j}.items))$
*By transitivity, then,*
*5.* $timestamp(head(s_{\rho_i}.items)) \leq timestamp(head(l_{\psi_k}.items))$

*Hence, the constraint rule is satisfied. Therefore, the conversion, K, as a whole respects eventqueue.* □

Again, this result matches our expectations. The buffer-overflowing program mentioned earlier got into trouble only because the programmer assumed that the event queue's buffer bound would not change, and therefore allocated a fixed buffer for receiving events from the queue. Programmers that do not assume that the event queue's buffer bound is fixed should allow for arbitrarily many events to be emptied from the queue, and thus avoid the error of the previous program.

# 5 Incorporating Concrete Types

So far we have discussed the *respects* relation in terms of abstract types since the subtype relation is defined in terms of a relation on abstract types. It makes sense, however, to consider the *respects* relation in terms of concrete types too. For example, when we implement an abstract type in a programming language, we choose a representation type for the abstract type and define the abstract type's methods in terms of methods on the representation type.

The TOM context introduces another kind of concrete type. When a user retrieves an object from a remote site, in reality that object is encoded in terms of some transmissible type, a concrete representation of the abstract object. These transmissible types are in turn represented in terms of primitive types that the underlying communication substrate understands; for TOM, and for the purposes of this paper, it suffices that every transmissible object be representable in terms of sequences of bytes; part of the byte sequence might represent metadata (e.g., the name of the abstract type) and the rest represents the data object itself.

Both of these kinds of concrete types may give rise to a new kind of converter, that from a concrete type to another concrete type. For example, in programming languages, if we have an abstract point type with two different representations, one using Cartesian coordinates and one using polar, we may want to implement a converter that takes any Cartesian point and produces the corresponding polar coordinates. Similarly, for an abstract matrix type, we may want to represent matrices in terms of both row-order and column-order and define converters between the two.

In the TOM context, integers may be represented in terms of a 32-bit sign extension byte sequence or a twos-complement, little-endian byte sequence, or even ASCII strings. These are all plausible concrete representations of integers and conversions between them should respect the abstract integer type.

To extend our definition of *respects* to accommodate converters from concrete type to concrete type, we borrow from the programming language community: we use the very same abstraction function used to prove the correctness of data representations first introduced by Hoare in 1974 [4].

Let converter $K : A_{conc} \rightarrow B_{conc}$ be defined on two concrete types $A_{conc}$ and $B_{conc}$. Then if $A_{conc}$ and $B_{conc}$ are correct implementations of (abstract) types $A$ and $B$, respectively, there exist abstraction functions:

$$\mathbf{A} : A_{conc} \rightarrow A$$
$$\mathbf{B} : B_{conc} \rightarrow B$$

We modify the definition of *respects* by modifying the definitions of $\alpha$ and $\beta$ of Section 3.1 accordingly. Assuming that $dom(\alpha_0) \subseteq ran(\mathbf{A})$ and $dom(\beta_0) \subseteq ran(\mathbf{B})$, we define

$$\alpha = \alpha_n \circ \ldots \circ \alpha_0 \circ \mathbf{A}$$
$$\beta = \beta_m \circ \ldots \circ \beta_0 \circ \mathbf{B}$$

That is, we first apply the abstraction function $\mathbf{A}$ on the concrete value of type $A_{conc}$ to form an abstract value of type $A$; we do the same to the concrete value of type $B_{conc}$ using $\mathbf{B}$. $K$ respects $T$ if the same condition holds as before, but using the revised $\alpha$ and $\beta$ abstraction functions defined above. In the case that the converter $K$ maps an abstract type to a concrete one or vice versa, then we can omit the application of $\mathbf{A}$ or $\mathbf{B}$ as appropriate.

# 6 Real World Examples

The first set of examples arose out of our work in building and using the TOM conversion service. The second is motivated by the imminent and infamous Y2K problem.

## 6.1 TOM Examples

In building TOM, we were faced with a choice of how to represent the directory type; one way is as a list of strings type. The list of strings type can in turn be represented by the sequence of bytes type, i.e., a transmissible type. Thus, a client who wishes to view the contents of a remote directory can do so even though the client's file system (e.g., Window NT) may differ from the server's (e.g., AFS or NFS).

In using TOM, typically, a client retrieves an object from a remote server. The client would like to view the object, originally in type $A$, as an object of type $B$ so far as it respects some type $T$. In the client's mind some abstract conversion from $A$ to $B$ is being performed. The problem is that the $A$ object has to be represented in terms of something transmissible across the wire; so first it is encoded into some transmissible type and then the client decodes the transmitted object into a $B$ object. Here $A_{conc}$ and $B_{conc}$ may very well be the same (transmissible) type, e.g., sequence of bytes, but at both the client and server sides abstract interpretations are defined on $A_{conc}$ and $B_{conc}$ to yield respectively objects of $A$ and $B$. For example, suppose a client fetches a compressed Word file from a Web server and wants to view it as an HTML file in a Web browser. Both the Word file and the HTML file are ultimately represented as sequences of bytes. The conversions required to uncompress and to convert the Word document to HTML respect a generic document type, via the abstraction that captures both their relations to the generic document type, and the relation of their byte-sequence representation to their abstract types Word and HTML.

Finally, consider the problem of parsing integers retrieved from a text document and storing them in a packed byte array. We can model this problem as a conversion from a string-based representation of integers to a byte-based representation. We have an abstract type int, a concrete type ibytes, and another concrete type istr. The abstraction function $\alpha$: istr $\rightarrow$ int is *atoi*, i.e., the standard C library function. The abstraction function $\beta$: ibytes $\rightarrow$ int is defined as follows for a given ibytes value $b$:

$$b[0] + 256 * b[1] + \ldots 256^{n-1} * b[n-1]$$

where $n$ is the number of bytes in $b$. Then $\beta^{-1} \circ \alpha$ is a conversion from istr to ibytes that respects int. This example illustrates a common way to find a conversion function: In general, if types $A$ and $B$ are "below" type $T$ by a composition of subtype and representation relations, and $\alpha$ is the abstraction function from $A$ to $T$ and $\beta$ is the abstraction function from $B$ to $T$, and $\beta$ is invertible, then $\beta^{-1} \circ \alpha$ is a conversion from $A$ to $B$ that respects $T$.

## 6.2 Y2K Examples

The data stored in many complex systems periodically needs to be converted to new formats. Some of these conversions are mandated by the passage of time, others by changing requirements and environments. The Y2K date conversion problem alone is estimated to cost billions of dollars to address, and it is only one of several conversion problems with hard deadlines [6].

The challenge of these conversions is not simply to change data formats appropriately, but also to ensure that the programs that use the data will continue to operate normally, or to upgrade them so that they will. Especially when faced with a conversion problem with a hard deadline, such as the Y2K problem, it is often impractical to convert at the same time all of the data and all of the code that operates on the data. Moreover, the desire for upward compatibility requires programs to be able to handle both the old and new data formats.

The conversion of data using two-digit years to data using four-digit years may require different changes in programs that use the data, depending on how two-digit years are interpreted. There are at least three distinct interpretations of two-digit years, and thus three distinct respectful type conversions.

Consider the date "4 July 99". In the simplest case (Figure 14), the year here is interpreted as the actual year modulo 100, with no assumption made about the century. (Two-digit years in historical letters are interpreted in this way, for example. It is also a reasonable interpretation for databases where the range of dates is known to be small.) In this interpretation, the four_digit_year type is an extension subtype [8] of the two_digit_year type. Any conversion from two_digit_years to four_digit_years that simply adds century information, e.g., as a new "century" attribute, will respect the two_digit_year supertype. As long as the program that operates on the two_digit_years does not make any assumptions about the centuries of the dates it reads, it will continue to operate correctly with four_digit_years, provided that it reads and writes dates through an interface that abstracts away representation details, e.g., an object-oriented one. (Here is an example of where $A$ and $T$ are the same type in the *respects* relation.)

A second common interpretation of two-digit years (Figure 15), particularly in older computer programs, is that they represent years between 1900 and 1999. Here a two_digit_year date type and a four_digit_year date type are both constrained subtypes [8] of a more general date type. For the date type to be a legal supertype of both the two_digit_year and four_digit_year date types, the supertype must allow the setting of a date's year to either succeed or fail, at least if they fall outside the 1900-1999 date range. The obvious conversion from the two_digit_year type to the four_digit_year type respects this supertype. In order to work correctly through such a conversion, programs will have to be prepared to deal with years outside the 1900-1999 range and also with failures when attempting to set the date outside this range.

A third interpretation is used in certain Unix-derived libraries (Figure 16), where the "year" field is interpreted as the actual year minus 1900. Here, the year in "4 July 99" is still interpreted as 1999, as in the second interpretation, but there is no constraint on the allowable years; a date in the year 2000, for instance, is represented, somewhat counterintuitively, as "4 July 100". In this third interpretration, the dates with

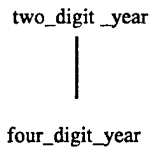

Figure 14: One Y2K Interpretation: A converter $K_1$ : two_digit_year → four_digit_year respects two_digit_year, where $K_1$ enhances the two_digit_year value with century information.

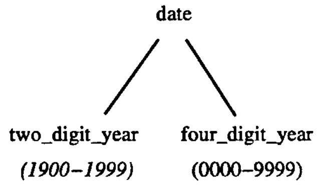

Figure 15: A Second Y2K Interpretation: A converter $K_2$ : two_digit_year → four_digit_year respects date, where $K_2$ adds 1900 to the two_digit_year value.

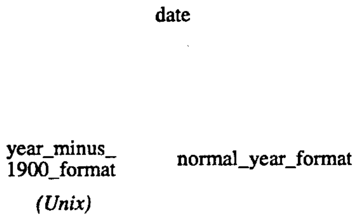

Figure 16: A Third Y2K Interpretation: Here, the dashed arrows denote the *representation* relation, not the subtype relation; year_minus_1900_format and normal_year_format are two different concrete types representing the same abstract date type. A converter $K_3$ : year_minus_1900_format → normal_year_format respects date, where $K_3$ adds 1900 to its Unix-like date.

compact year representations, and those with the year fully spelled out, as in "4 July 1999", are simply two different concrete representations of the same abstract date type. Conversions from one to the other respect the abstract type, and programs can easily use either representation, again as long as they read and write dates through an abstract interface that hides the exact representation details.

Unfortunately, many legacy programs may interpret dates inconsistently. Perhaps some modules use the first interpretation of the dates above, while others use the second. In such a case, there may be no one respectful conversion from two-digit years to four-digit years, because the two-digit years are not consistently interpreted as the same type, with the same semantics. However, making this discovery can itself help in adapting such a program for a Y2K conversion. Once we attempt to determine the types of objects to be converted and discover that there are two or more slightly inconsistent types used, we can fix some of the modules so that they all interpret types consistently. Alternatively, we can model the overall conversion as two conversions from two slightly different types to one common type. We can then analyze both conversions to see what abstract types they respect, and use this information to help ensure that the programs will continue to behave properly after a conversion.

# 7 Related Work

There are notions of "respectful" conversions that are stronger or weaker than the one we present in this paper; they may be more appropriate in certain situations.

In earlier work [11], we gave a simpler model of respectful conversion for immutable types. In that model, when an object of type $A$ is converted to an object of type $B$ using a conversion that respects $T$, the objects $A$ and $B$ cannot be distinguished from type $T$'s viewpoint. That is, the conversion preserves the information specified for $T$. Since the objects are immutable, there are no mutators or history constraints to consider. We showed in that paper how that simpler model is useful for applications that retrieve and analyze data. When applied to mutable objects, however, the failure to consider history constraints may produce behavior in the converted object that is inconsistent with the past behavior of the original object, as we saw in our queue example. When the definition of respects given in this paper is applied to immutable objects, the predicates under the **Methods** condition to determine respectful conversions simplify to the predicates given in our previous paper; the **Constraint** condition is entirely unnecessary. Also, the the PNG and GIF example presented in the earlier paper considered a simpler version where the image types are immutable; the eventqueue and Y2K examples in this paper are entirely new.

Applications that pass converted data back and forth between heterogeneous programs may require stronger guarantees on conversions. For example, the Mockingbird system [1], defines a notion of *interconvertibility* where data conversions are fully invertible. With this policy, data converted to another format can always be converted back without loss of information. This concept corresponds in our model to a conversion between two representations of some type $T$, where the conversion respects type $T$. While interconvertibility makes it easy to exchange transformed data without risk of losing information, it is often too strong a constraint on conversions. Our model of respectful type conversions is more flexible. In respectful conversions, some information can be lost or changed in the conversion, provided that the information and behavior of the respected type is preserved.

# 8 Summary and Conclusions

In this paper, we extended our earlier definition of a novel notion of *respectful type converters* to capture what behavior a conversion function preserves when transforming objects of one type to another; our extension deals with mutable types. We greatly leverage off the Liskov and Wing notion of behavioral subtyping to characterize this information succinctly. Their framework gives us the key technical tool we need; in particular, the abstraction functions, $\alpha_i$, define formally how two different objects can be viewed as the same. We also can easily incorporate concrete types by leveraging off Hoare's abstraction functions for proving the correctness of the implementations of abstract types.

We illustrated the use of our definitions on a type family for images and on one for event queues. We further motivated the appeal of respectful type converters in the context of the TOM conversion service and the Y2K problem.

As discussed for the Y2K examples, analyzing the types that a conversion respects allows developers to determine where programs will continue to behave normally after data is converted, and where they may behave unexpectedly or erroneously. Intuitively, if a conversion respects a type $T$, then after an object of type $A$ is converted to an object of type $B$ in a conversion that respects $T$, programs that operate on the objects using the interface and expectations of $T$ will encounter no surprises. Programs that use more detailed interfaces or that rely on behavioral assumptions specified by $A$ or $B$ but not by $T$, however, may encounter problems. Reviewing the assumptions programs make about data and seeing what types conversions respect allow us to detect possible conflicts introduced by converted data, and to adjust programs appropriately.

This spin on the Y2K problem suggests that when faced with a massive conversion task, we should consider all three points of the triangle, A, B, and T, where A in the obsolete format of data, B is the new format, and T is the set of assumptions which the program continues to rely on despite a conversion of data from A to B.

## References

[1] Joshua Auerbach and Mark C. Chu-Carroll. The Mockingbird System: A Compiler-based Approach to Maximally Interoperable Distributed Programming. Technical Report RC 20718, IBM, Yorktown Heights, NY, 1997.

[2] Wayne Brookes and Jadwiga Indulska. A Type Management System for Open Distributed Processing. Technical Report 285, University of Queensland, St. Lucia, Qld., Australia, 1994.

[3] S.J. Garland and J.V. Guttag. An Overview of LP, the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 137–151, Chapel Hill, NC, April 1989. Lecture Notes in Computer Science 355.

[4] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(1):271–281, 1972.

[5] J.J. Horning, J.V. with S.J. Garland Guttag, K.D. Jones, A. Modet, and J.M. Wing. *Larch : Languages and Tools for Formal Specification.* Springer-Verlag, New York, 1993.

[6] Capers Jones. Bad Days for Software. *IEEE Spectrum*, 35(9):47–52, September 1998.

[7] Barbara Liskov. Data Abstraction and Hierarchy. In *OOPSLA '87: Addendum to the Proceedings*, 1987.

[8] Barbara Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM TOPLAS*, 16(6):1811–1841, November 1994.

[9] John Ockerbloom. Mediating Among Diverse Data Formats. Technical Report CMU-CS-98-102, Carnegie Mellon Computer Science Department, Pittsburgh, PA, January 1998. Ph.D. Thesis.

[10] Steve Putz. Design and Implementation of the System 33 Document Service. Technical Report P93-00112, Xerox PARC, Palo Alto, CA, 1993.

[11] Jeannette M. Wing and John Ockerbloom. Respectful Type Converters. *IEEE Transactions on Software Engineering*, 1999. to appear.

[12] Amy M. Zaremski. Signature and Specification Matching. Technical Report CS-CMU-96-103, CMU Computer Science Department, January 1996. Ph.D. Thesis.

# Appendix A: Larch Traits and Type Specifications

This appendix contains the following Larch specifications: Color trait for color literals, ColorSet trait for sets of colors, Frame trait, FrameSeq trait, GIFImage trait, Gammas trait, PNGImage trait, PixelMap trait, Event trait, event type, and EventQueue trait. Appendix A of the Larch Book [5] contains traits for Boolean, Integer, FloatingPoint, Set, Deque, Array2, TotalOrder, and Queue, all of which we use below.

```
ColorLiterals: trait
  % A trait for N colors where BLACK = 0 and WHITE = 1 and N >> 256.
    Color enumeration of BLACK, WHITE, 2, ..., N-1
end ColorLiterals

ColorSet(Color, CS): trait
  includes ColorLiterals, Set (Color, CS)
end ColorSet

 Frame(F): trait
  includes Array2 (Color, Integer, Integer, F), ColorSet (Color, CS)
  introduces
    $xmin, xmax, ymin, ymax$ : F $\rightarrow$ Integer
    $colorrange$ : F $\rightarrow$ CS
    $inframe$ : F, Integer, Integer $\rightarrow$ Boolean
  asserts for all $i, j$ : Integer, $f$ : F
    $xmin(f) \leq xmax(f)$
    $ymin(f) \leq ymax(f)$
    $inframe(f,i,j) = (xmin(f) \leq i \leq xmax(f)) \wedge (ymin(f) \leq j \leq ymax(f))$
    $inframe(f,i,j) \Rightarrow f[i,j] \in colorrange(f)$
end Frame

 FrameSeq(F, FS): trait
  includes Deque (Frame, FS)
  introduces
    $overlay$: FS, Integer, Integer $\rightarrow$ Color
    $changepixel$: FS, Integer, Integer, Color $\rightarrow$ FS
    $colorrange$: FS $\rightarrow$ CS
    __[__]: FS, Integer $\rightarrow$ F
  asserts for all $i, j, k, l$: Integer, $c$: Color, $f$: F, $fs$: FS
    $overlay(fs, i, j)$ = if $len(fs) = 0$ then BLACK else
          if     $inframe(last(fs), i, j)$
          then   $last(fs)[i, j]$
          else   $overlay(init(fs), i, j)$
    $colorrange(empty) = \{\}$
    $colorrange(fs \vdash f) = colorrange(fs) \cup colorrange(f)$
    $(fs \vdash f)[i]$ = if $i = len(fs \vdash f)$ then $f$ else $fs[i]$
    $overlay(changepixel(fs,i,j,c),k,l)$ = if $i = k \wedge j = l$ then $c$ else $overlay(fs,k,l)$
exempting
  $\forall i : Integer \, . \, empty[i]$
  $\forall i \leq 0 \, . \, fs[i]$
  $\forall i \geq len(fs) \, . \, fs[i]$
end FrameSeq

GIFImage: trait
  includes FrameSeq (G for FS), ColorSet(Color, CS)
  asserts for all $g$: G
```

```
    BLACK ∈ colorrange(g)
end GIFImage

Gammas: trait
  includes FloatingPoint (Gamma for F)
  introduces
    STDG: → Gamma
    gc : Color, Gamma, Gamma → Color
  asserts for all c: Color, g, h, i: Gamma
    gc(c, g, g) = c                      "reflexivity"
    gc(gc(c, g, h), h, i) = gc(c, g, i)  "transitivity"
end Gammas

PNGImage: trait
  includes Frame (P for F), Gammas
  introduces
    gamma: P → Gamma
    same_bounds_and_gamma: P, P → Boolean
  asserts for all p, q: P
    same_bounds_and_gamma(p, q) = (gamma(p) = gamma(q)∧
      xmin(p) = xmin(q) ∧ xmax(p) = xmax(q) ∧ ymin(p) = ymin(q) ∧ ymax(p) = ymax(q))
end PNGImage

PixelMap: trait
  includes Array2 (Color, Integer, Integer, PM), ColorSet (Color, CS)
  introduces
    colorrange: PM → CS
  asserts for all i, j: Integer, pm: PM
    BLACK ∈ colorrange(pm)
    pm[i, j] ∈ colorrange(pm)
end PixelMap

Event: trait
  includes TotalOrder (Time for T)
  introduces
    timestamp: Ev → Time

event: type
  uses Event (event for Ev)
  end event

EventQueue: trait
  includes Queue (Ev for E), Event
    Q tuple of items: C, bound: Integer
```

# Appendix B: Proof of Claim in Section 4.1

At the end of Section 4.1 we claimed that from pixel_map's perspective the future subhistories of an unconverted GIF image will be the same as those of the converted PNG image. We give a more formal argument here.

First, we need to define a few terms. Let $Vals : Obj \times Val \times Method* \rightarrow 2^{Val}$, such that $Vals(o, v, \bar{m})$ is the set of possible values that object $o$ can have after applying the sequence of method calls in $\bar{m}$ on $o$ starting with value $v$. This set may contain more than one element because method specifications are in general nondeterministic.

**Defn 1 Observational Equivalence for Values** ($v_1 =_T v_2$) *Two values $v_1$ and $v_2$ are* observationally equivalent *with respect to a type $T$ if:*

*For each method $m$ of $T$, and for all objects $y : T$ and $z : T$, for all subcomputations $y_{pre}\ m\ y_{post}$ and $z_{pre}\ m\ z_{post}$ such that $y_{pre} = v_1$ and $z_{pre} = v_2$:*

1. *$m.pre[y_{pre}/x_{pre}] \Leftrightarrow m.pre[z_{pre}/x_{pre}]$ and*
2. *$m.post[y_{pre}/x_{pre}, y_{post}/x_{post}] \Leftrightarrow m.post[z_{pre}/x_{pre}, z_{post}/x_{post}]$*

These conditions are analogous to the two **Methods** rules of the definition of the respects relation, but are simpler since they relate values of the same type $T$. Note that if $v_1 = v_2$ then trivially $v_1 =_T v_2$.

The above concept becomes more interesting when we work with values of different types, e.g., $A$ and $B$. We extend observational equivalence in two ways at once: to work on *sets* of values of *different types*:

**Defn 2 Observational Equivalence for Value Sets** ($S_1 \equiv_T S_2$) . *Let $S_1$ be a set of values of objects of type $A$, $S_2$ a set of values of objects of type $B$, and $\alpha : A \rightarrow T$ and $\beta : B \rightarrow T$ be the respective abstraction functions for showing that $A$ and $B$ are subtypes of $T$. Then $S_1 \equiv_T S_2$ iff*
$\forall v_1 \in S_1 . \exists v_2 \in S_2 \ .\ \alpha(v_1) =_T \beta(v_2) \ \wedge\ \forall v_2 \in S_2 . \exists v_1 \in S_1 \ .\ \beta(v_2) =_T \alpha(v_1)$

Informally, if every element in $S_1$ has a corresponding element in $S_2$ that looks the same from $T$'s standpoint, and vice versa, then $S_1$ and $S_2$ are observationally equivalent with respect to $T$. Notice that we do not require that $v_2 = K(v_1)$ in the above definition.

Our claim is that, when a GIF is converted to a PNG image, that PNG image has the same possible future subhistories from pixel_map's point of view that the original GIF has. Formally,

**Claim 6** *Given a GIF object $g$ with value $v$, and a PNG object $p$ with value $K(v)$, where $K$ is the GIF to PNG converter that respects pixel_map as given in Section 4.1, then for all sequences $\bar{m}$ of pixel_map method calls on objects $g$ and $p$, $Vals(g, v, \bar{m}) \equiv_T Vals(p, K(v), \bar{m})$.*

*Proof: By induction on the length of $\bar{m}$.*

*Base case: $len(\bar{m}) = 0$. If no methods have been called on $g$ and $p$, then their values have not changed, so $Vals(g, v, \bar{m}) = \{v\}$ and $Vals(p, K(v), \bar{m}) = \{K(v)\}$. We have already shown in Claim 3 that $\alpha(v) = \beta(K(v))$ and thus trivially $\alpha(v) =_T \beta(K(v))$. Thus, $\{v\} \equiv_T \{K(v)\}$, and thus $Vals(g, v, \bar{m}) \equiv_T Vals(p, K(v), \bar{m})$.*

*Inductive case: Assume the claim holds for $\bar{m}$ where* $\mathrm{len}(\bar{m}) > 0$. *Show it holds for $\bar{m} \vdash m$, for some method call $m$. That is,*

*IH: $Vals(g, v, \bar{m}) \equiv_T Vals(p, K(v), \bar{m})$.*

*We need to show*

$$Vals(g, v, \bar{m} \vdash m) \equiv_T Vals(p, K(v), \bar{m} \vdash m).$$

*Case 1. $m$ is a call to a non-mutator. Since $m$ does change any object values, $Vals(g, v, \bar{m} \vdash m) = Vals(g, v, \bar{m})$. Similarly, $Vals(p, K(v), \bar{m} \vdash m) = Vals(p, K(v), \bar{m})$. Thus, by IH, $Vals(g, v, \bar{m} \vdash m) \equiv_T Vals(p, K(v), \bar{m} \vdash m)$.*

*Case 2. $m$ is a call to a mutator. The only mutator method in pixel_map is* set_color. *It can either fail or succeed. Because it is always allowed to fail, and thus have no effect, we know $Vals(g, v, \bar{m}) \subseteq Vals(g, v, \bar{m} \vdash m)$ and $Vals(p, K(v), \bar{(m)}) \subseteq Vals(p, K(v), \bar{m} \vdash m)$.*

*If it succeeds, it can also have an effect. In this case, for any* $v_1 \in Vals(g, v, \bar{m})$ *choose* $v_2 \in Vals(p, K(v), \bar{m})$ *such that* $\alpha(v_1) =_T \beta(v_2)$. *Our induction hypothesis tells us that* $v_2$ *exists. (The parallel argument holds when choosing some* $v_1$ *for any* $v_2$*.) Let* $v'_1$ *and* $v'_2$ *be the respective new values of* $g$ *and* $p$ *after the successful call to* set_color. *We need to show that* $\alpha(v'_1) =_T \beta(v'_2)$. *The only observer in pixel_map is* get_color. *It has a trivial pre-condition so the first part of Definition 1 trivially holds. From* get_color*'s post-condition, we need to show that* $result = \alpha(v'_1)[i,j] \Leftrightarrow result = \beta(v'_2)[i,j])$

*The result of a successful call to* set_color(x, y, c) *on* $g$*, no matter what the previous color was, will be the same as for* $p$*. That is,* $c = \alpha(v'_1)[x,y] \wedge c = \beta(v'_2)[x,y]$. *For other coordinates,* $i \neq x$ *or* $j \neq y$*, the result of calling* get_color(i, j) *is the same as it was before this last call* $m$ *to* set_color. *That is,* $\alpha(v'_1)[i,j] = \alpha(v_1)[i,j] \wedge \beta(v'_2)[i,j] = \beta(v'_2)[i,j]$.

*Thus,*

1. $\forall i, j \ . \ \alpha(v'_1)[i,j] = \beta(v'_2)[i,j]$

*Thus,*

2. $\alpha(v'_1) =_T \beta(v'_2)$

*Thus,*

3. $Vals(g, v, \bar{m}) \cup \{v'_1\} \equiv_T Vals(p, K(v), \bar{m}) \cup \{v'_2\}$

*Thus,*

4. $Vals(g, v, \bar{m} \vdash m) \equiv_T Vals(p, K(v), \bar{m} \vdash m)$

□